Express Mail Label
No. EF230793936US

MIC-13

PROGRAM LOADING MECHANISM THROUGH A SINGLE INPUT DATA PATH

Background of the Invention

This invention relates to the loading of program code, data, and control information into a processing engine. More particularly, this invention relates to loading program code, data, and control information into a processing engine through a single data path.

A processing engine processes input data to generate output data. A processing engine typically includes the following: a memory, a program counter, control logic, an execution pipeline, and a register file. The memory holds a stored program, which is a set of instructions stored in a block of memory. The program counter typically contains an address to an instruction in memory. After an instruction is read, the program counter is reloaded with the address of the next instruction to be read. Control logic decodes the current instruction, thereby controlling the execution pipeline. An instruction typically contains the following: one or more operands (e.g., contents of source registers or memory addresses, or constants), an operation code (e.g., add, subtract, multiply, shift, load, store, branch, etc.), and a destination register or memory address for a resulting value. The execution pipeline computes output data from input data by performing arithmetic and logic operations defined by

the decoded instruction. Each operation is preferably processed in multiple stages (e.g., a fetch stage, a decode stage, an execute stage, and a write-back stage) such that, where possible, the different stages can be overlapped to increase throughput (i.e., the rate at which temporary values or partial results are computed by the execution pipeline). The register file typically holds temporary values or partial results computed by the execution pipeline. In addition, the register file can hold constants that are initialized before the program executes. The temporary values and constants can be changed from time to time during program execution.

Processing engines of a known class typically do not have the ability to perform random access of their input data. Instead, those engines can only access (or read) a next piece of data from the input data stream. If no data is available, such processing engines stall until data becomes available. Logically, the input data stream can be considered the output of a first-in-first-out (FIFO) system. The data positioned first in the input data stream is processed first. Successive sequences of input data may need to be processed by different programs or by using different sets of constants in the register file. However, the memory and register file are often too small to accommodate all possible programs and constants simultaneously. As a result, either the memory, register file, or both will need to be reloaded in preparation for each sequence of input data.

A typical sequence of operations required by known processing engines is shown in table (1).

1	Load a program into Memory
2	Load constants into the Register File
3	Initialize the Program Counter
4	Supply an arbitrary number of pieces of input data
5	Wait for all of the input data to be processed
6	Load new constants into the Register File
7	Initialize the Program Counter
8	Supply an arbitrary number of pieces of input data
9	Wait for all of the input data to be processed
10	Load a new program into Memory
11	Initialize the Program Counter
...	...

Table (1)

Different data paths are used for initializing the system and executing a stored program. To ensure that a data stream has been processed to completion before loading a new program or a new set of constants, initialization must be synchronized by external control logic.

Known processing engines have several disadvantages. First, the use of multiple input data paths increases hardware complexity. Additional wiring and logic circuitry are needed. For example, multiplexers are needed to select whether initialization data or program data is to be sent to a register file. Also, multiple operations (e.g., run mode and setup mode) do not implement the same functions the same way. For example, the run mode (i.e., program mode) function of writing to a register file is typically different than the setup mode function of writing to a register file. In setup mode, an address is sent to a register file directly from an

initialization path and a multiplexer. In run mode, an address is sent to the register file via control logic and the multiplexer. Second, the use of external control logic increases hardware and creates a dependency on an external source. External control logic is needed to synchronize the loading and execution of data through the multiple input data paths such that data is processed in the correct order at the appropriate time. The processing engine must function in the same timing domain as the external control logic -- the processing engine cannot operate in a separate timing domain nor can data be supplied to it at a sporadic rate.

In view of the foregoing, it would be desirable to provide a processing engine with an efficient mechanism for loading program code, data, and control information.

It would also be desirable to provide a processing engine that allows a program to be processed with little or no external control logic.

Summary of the Invention

It is an object of this invention to provide a processing engine with an efficient mechanism for loading program code, data, and control information.

It is another object of this invention to provide a processing engine that allows a program to be processed with little or no external control logic.

In accordance with this invention, a processing engine is presented with a single input data path. Program data and setup data are passed through the same input data path. To identify the input data as either setup data or program data, each piece of input data preferably contains one or more additional bits, known as identification bits.

The identification bits indicate whether the processing engine is in setup mode (associated with setup data) or run mode (associated with program data). Initially after a reset, the processing engine typically runs in setup mode. In setup mode, control logic executes a setup program, which may make use of instructions and operations that are not typically available in run mode. In run mode, the processing engine executes instructions that are read from memory. When the input data changes from setup data to program data, the processing engine automatically switches from setup mode to run mode. Similarly, when the input data changes from program data to setup data, the processing engine automatically switches from run mode to setup mode. When input data is not available, the system enters a wait state in its current mode.

Input data is passed through a single input data path into an execution pipeline. A piece of input data identified as program data is processed in the execution pipeline under the control of the current instruction that has been read from memory. A piece of input data identified as setup data is propagated through the execution pipeline via execution of a pass-through instruction. When setup data reaches the output data bus, control logic generates appropriate control signals to load the data into appropriate storage locations. Setup data can be decoded further from either additional identification bits or some subset of the input data. Any additional information supplied with a piece of input data can be part of an instruction sequence of the setup program.

Brief Description of the Drawings

The above and other objects and advantages of the invention will be apparent upon consideration of the following detailed description, taken in

conjunction with the accompanying drawings, in which like reference characters refer to like parts throughout, and in which:

FIG. 1 is a block diagram of a known multiple input data path processing engine;

FIG. 2 is a block diagram of a single input data path processing engine in accordance with the present invention;

FIG. 3 is an illustration of a sample input data stream in accordance with the present invention; and

FIG. 4 is a flow diagram illustrating the processing of program and setup data in a single input data path processing engine in accordance with the present invention.

Detailed Description of the Invention

The present invention provides a processing engine that loads program code, data, and control information through a single input data path.

FIG. 1 shows a known multiple input data path processing engine 100 that has initialization paths 104, 106, 108, 110, and 112 for setup data and an input data path 102 for program data. These initialization paths are controlled by external control logic, which is not shown. A program can be loaded into a memory 122 via an initialization input data path (D_{IN}) 108 at an address specified from an initialization write address path (W_RA_{DDR}) 110. Program instructions are typically stored sequentially in a block of memory 122. Program execution can be started by loading a memory address of a first program instruction into a program counter (PC) 126 via initialization path 112, control logic 124, and path 150. Data (such as program constants) can be loaded into a register file 114 via initialization path 104, a

multiplexer 116, and an input data path (D_{IN}) 132. An address of register file 114 to which data is to be written can be specified using initialization path 106, a multiplexer 120, and a write address path (W_RA_{DDR}) 138.

Once memory 122, register file 114, and program counter 126 are initialized, input data from path 102 is processed in an execution pipeline 118. An instruction is fetched from memory 122 at a current address of program counter 126 via path 146. This program counter value is also sent to control logic 124 via read address path (R_DA_{DDR}) 146. Control logic 124 generates a new program counter value (by incrementing the current value or by calculating a branch destination) and sends the new value to program counter 126 via path 150. The fetched instruction is sent to control logic 124 via output data path (D_{OUT}) 144, where the instruction is decoded to generate control signals for execution pipeline 118 via path 140. Control logic 124 can send a read address to register file 114 via a read address path (R_DA_{DDR}) 136. This may be used to read a register value for use in execution pipeline 118. Control logic 124 can also send a write address to register file 114 via a path 142, multiplexer 120, and write address path (W_RA_{DDR}) 138. This may be used to write an output from execution pipeline 118 into register file 114. Data from register file 114 can be sent to execution pipeline 118 via an output data path (D_{OUT}) 130. Once the instruction has been processed, its result can be sent to an output data path 128. Data path 128 is used to transfer processed data out of processing engine 100. When control logic 124 determines that the current instruction will generate a processed data output, it signals this using an output data valid signal 152. Data path 128 is also used to transfer temporary or partially-processed data to register

file 114 via a path 134, multiplexer 116, and input data path (D_{IN}) 132.

FIG. 2 shows a single input data path processing engine 200 in accordance with an embodiment of the present invention. Instead of an input data path for program data and a plurality of separate initialization paths for setup data as shown in FIG. 1, processing engine 200 uses a single input data path 202. Each piece of input data from input data path 202 includes additional identification bits. These additional identification bits are separated from the rest of the input data and transported along input data identification (ID) path 204. Although not shown, control logic can be used to read the identification bits (e.g., reading the most significant bits) and sending the identification bits to input data ID path 204. The identification bits may be one or more bits. They indicate whether a given piece of input data is setup data or program data, and may also contain other information related to that piece of input data.

When the identification bits indicate that data is program data, control logic 124 signals execution pipeline 118 via path 140 to process the next instruction of a stored program residing in memory 122 (via output data path (D_{OUT}) 144, control logic 124, and path 140). The instruction may be executed using any suitable pipelining method. Execution pipeline 118 continues to execute instructions from the stored program (which may include reading values from register file 114 or writing values to the output of processing engine 200 via output data path 128) until control logic 124 reads the identification bit(s) of the next piece of data. The identification bit(s) of the next piece of data determines what mode processing engine 200 will operate in next.

When the identification bits indicate that data is setup data, control logic 124 signals execution pipeline 118 via path 140 to execute a pass-through instruction. A pass-through instruction allows a piece of input data to propagate through to output data path 128 without modification. The setup data will be passed to output data path 128. Control logic 124 may perform additional decoding of the setup data either via path 204 or via path 210.

Register file 114 can be loaded with setup data or program data. In both cases, register file 114 receives input data directly from output data path 128 via input data path (D_{IN}) 206. A register file 114 address to which input data can be written is sent from control logic 124 via write address path (W_RA_{DDR}) 138. Data in register file 114 can be read under the control of a stored program: output data is driven into execution pipeline 118 via output data path (D_{OUT}) 130. A register file 114 address from which output data can be read is sent from control logic 124 via a read address path (R_DA_{DDR}) 208.

Memory 122 can receive data from output data path 128 via path 210. For example, setup data may cause a program to be loaded into memory 122. Memory 122 can also connect to program counter 126 via path 146. Path 146 can be used as a read address path (R_DA_{DDR}) (e.g., during run mode) or a write address path (W_RA_{DDR}) (e.g., during setup mode). Data can be sent from memory 122 to control logic 124 via output data path (D_{OUT}) 144.

Program counter 126 is preferably used to access appropriate locations in memory 122. A value in program counter 126 can be sent to memory 122 via path 146. The value in program counter 126 is either incremented by or replaced with a new counter address from control logic 124 via data path 150.

FIG. 3 illustrates a sample input data stream 300. Input data stream 300 includes multiple pieces of input data 312. Each piece of input data 312 preferably contains an identification field 302, which is one or more bits, and a data field 306, which contains either program data or setup data. An identification bit of "0" may indicate, for example, setup data (setup mode), while an identification bit of "1" may indicate program data (run mode).

A piece of input data that is identified as setup data can be decoded further either from additional identification bits 302 or from some subset of the input data itself. This allows each piece of input data to contain both a setup data value and a setup program instruction. The sequence of input data values that are identified as setup data act as the instruction sequence of the setup program, which is executed by the control logic in setup mode.

The instructions of the setup program perform several important functions. One function is loading a program counter (PC) with a program memory address. The PC value is preferably supplied on the input data path. A second function is loading memory. A set of instructions are supplied on the input data path, and each instruction is loaded into memory at the current PC value. The PC is then incremented before a next instruction is loaded. A third function is loading a register file counter. The counter value can be supplied on the input data path. A fourth function is loading the register file. A value supplied on the input data path is loaded into the register file at the current value of the register file counter. The register file counter is then incremented before a next value is loaded.

When any setup instruction is executed by control logic, the associated input data is propagated

through the execution pipeline via execution of a "pass-through" instruction. When the data reaches the output data path (which is the same data as that on the input data path), control logic generates the appropriate control signals to load the data in the appropriate storage location. While the system is in setup mode, it may be necessary to inhibit exceptions, such as interrupts, because the system may be in a transitory state and therefore unable to process them.

FIG. 4 illustrates the processing of program and setup data in a single input data path processing engine. Process 400 begins after the system has been reset. Because the "setup program" has little or no dependence upon the state of the processing engine at the time the setup program begins to execute, it can be used to set the initial state of the system. At step 404, the system enters setup mode. At step 406, process 400 determines whether a piece of input data is available, and if so, checks the identification bit. When the identification bit indicates setup data, process 400 moves to step 408 to process that piece of input data. When the piece of input data has been processed, process 400 moves back to step 406 to check for a next piece of input data. If no data is detected, process 400 moves to step 410, where the system waits in its current mode until a next piece of data is available.

When the identification bit indicates program data, process 400 moves to step 412 where the system enters run mode. Process 400 then moves to step 414 where it decodes a stored instruction at the current value of the program counter (PC). At step 416, process 400 determines if the decoded instruction requires any input data. If the instruction does not require input data, process 400 moves to step 422, where the processing engine executes the instruction

and updates the PC (but does not process any input data). If the execution of an instruction requires input data, process 400 moves to step 418. At step 418, process 400 examines the identification bit of a next piece of input data. If the piece of input data is program data, process 400 moves to step 422 where the instruction is executed, the PC is updated, and the input data is processed. Process 400 then moves back to step 414. If data is not available, process 400 moves to step 420 where the system waits in its current mode. If the identification bit indicates setup data, process 400 moves back to step 404 without executing the decoded instruction.

Thus, process 400 can run continuously, with the processing engine automatically switching between setup mode and run mode each time a change in identification bits is detected. In run mode, the processing engine executes instructions that are read from memory. In setup mode, control logic executes the setup program, which may make use of instructions and operations not available in run mode.

The single input data path processing engine of the present invention has several advantages. Foremost, initialization information passes through a single data path. All programs, constants, and program data can be pre-processed into a single data stream, requiring no external control logic to manage the flow of data. Arranging the input data into a single data stream allows self-synchronization. A new program will not be loaded until all the data from a previous program (which precedes it in the data stream) has been processed by that previous program. Input data is analogous to the output of a first-in-first-out (FIFO) system. The source of the input data advantageously can be in a separate timing domain or can supply data at a sporadic rate without compromising the

synchronization between the different pieces of program data and setup data in the input data stream. Both control logic and data paths are simplified by reusing functions that exist for normal operation. For instance, control logic already has the ability to write to the register file and the program counter. Because data required for state changes passes down the execution pipeline, it remains ordered with respect to program data. This allows state changes to be pipelined, avoiding the inefficiency of draining the execution pipeline. Finally, the instructions of the setup program advantageously do not reduce the opcode space available to normal programs because these instructions are only available in setup mode. Because setup data and program data are sent to the execution pipeline as separate pieces of input data, the number of bits representing setup data and program data can be customized.

Additional features or variations of the basic processing engine are also included in other embodiments of the present invention. For example, instead of reserving an identification field in the input data, one or more values that cannot normally occur in input data can be decoded and used to initiate the transition from run mode to setup mode. The same or a different value can be decoded and used to initiate the transition from setup mode to run mode.

Also, instead of the described setup program which can include a number of different instructions along with the input data, a simple system according to the invention may require only a single instruction or a single inflexible sequence. The mode change from run mode to setup mode can be sufficient to trigger the execution of a "hard wired" instruction or setup program by the control logic.

Alternatively, a system may require a highly complex setup program. In this case, the program according to the present invention can be stored in a separate read-only memory (ROM) and executed under the control of a dedicated setup program counter. This operation is similar to the behavior of a multi-threaded processor.

A "setup program" instruction can also be provided that propagates an input value straight through the execution pipeline to the output data path. This requires the addition of output data ID signals generated by control logic in accordance with the current operation mode. This allows appropriately tagged values to exist in the input data stream arbitrarily interleaved with program data. These special data values reach the output in a way that is asynchronous (relative to program execution), but have a pre-defined ordering relative to computed output values. Additional outputs from control logic, which are valid at the same time as the output data valid signal, can indicate whether a piece of output data is a value generated by the execution pipeline or propagated through the pipeline by a setup program.

Thus it is seen that a processing engine is provided for loading code, data, and control information through a single input data path with less complex hardware. One skilled in the art will appreciate that the present invention can be practiced by other than the described embodiments, which are presented for purposes of illustration and not of limitation, and the present invention is limited only by the claims which follow.